9. The Connection Library

Created: April 1, 2003
Updated: September 16, 2003

Summary

Connection Library [Library `xconnect`: include | src]

Includes a generic socket interface (`SOCK`), connection object (`CONN`), and specialized connector constructors (for sockets, files, HTTP and services), to be used as engines for connections. Also, provides access to load-balancing daemon and NCBI named service dispatching facilities.

While the core of the Connection Library is written in C and has an underlying C interface, the analogous C++ interfaces have been built to provide objects that work smoothly with the rest of the Toolkit.

i. Debugging Tools and Troubleshooting Documentation

ii. C++ Interfaces to the Library

- CONN-Based C++ Streams and Stream Buffers ncbi_conn_stream[.hpp | .cpp], ncbi_conn_streambuf[.hpp | .cpp]
- Diagnostic Handler for E-Mailing Logs email_diag_handler[.hpp | .cpp]
- Using the CONNECT Library with the C++ Toolkit ncbi_core_cxx[.hpp | .cpp]
- Multithreaded Network Server Framework threaded_server[.hpp | .cpp]

iii. Basic Types and Functionality (for Registry, Logging and MT Locks) ncbi_core[.h | .c], ncbi_types[.h]

iv. Portable TCP/IP Socket Interface ncbi_socket[.h | .c]

v. Connections and CONNECTORs

- Open and Manage Connections to an Abstract I/O ncbi_connection[.h | .c]
- Implement CONNECTOR for a ...
  - Abstract I/O ncbi_connector[.h | .c]
  - Network Socket ncbi_socket_connector[.h | .c]
  - FILE Stream ncbi_file_connector[.h | .c]

- HTTP-based Network Connection ncbi_http_connector[.h | .c]
- Named NCBI Service ncbi_service_connector[.h | .c]
- In-memory CONNECTOR ncbi_memory_connector[.h | .c]

vi. Servers and Services

- NCBI Server Meta-Address Info ncbi_server_info[.h | p.h | .c]
- Resolve NCBI Service Name to the Server Meta-Address ncbi_service[.h | p.h | .c]
- Resolve NCBI Service Name to the Server Meta-Address using NCBI Network Dispatcher (DISPD) ncbi_service[p_dispd.h | _dispd.c]
- Resolve NCBI Service Name to the Server Meta-Address using NCBI Load-Balancing Service Mapper (LBSM) ncbi_service[p_lbsmd.h | _lbsmd.c | _lbsmd_stub.c]
- NCBI LBSM client-server data exchange API ncbi_lbsm[.h | .c]
- Implementation of LBSM Using SYSV IPC (shared memory and semaphores) ncbi_lbsm_ipc[.h | .c]

vii. Memory Management

- Memory-Resident FIFO Storage Area ncbi_buffer[.h | .c]
- Simple Heap Manager With Primitive Garbage Collection ncbi_heapmgr[.h | .c]

viii. Connection Library Utilities

- Connection Utilities ncbi_connutil[.h | .c]
- Send Mail (in accordance with RFC821 [protocol] and RFC822 [headers]) ncbi_sendmail[.h | .c]
- Auxiliary (optional) Code for *ncbi_core.[ch]* ncbi_util[.h | .c]
- Non-ANSI, Widely Used Functions ncbi_ansi_ext[.h | .c]

daemons [src/connect/daemons]

- LBSMD
- DISPD

- Firewall Daemon

Test Cases [src/connect/test]

Note: Due to security issues, not all links in the public version of this file could be accessible by outside NCBI users. Unrestricted version of this document is available to inside NCBI users at: http://intranet.ncbi.nlm.nih.gov/ieb/ToolBox/CPP_DOC/libs/conn.html.

Contents

Overview

NCBI C++ platform-independent connection library (*src/connect* and *include/connect*) consists of 2 parts:

1. Lower-level library written in C (also used as a replacement of existing connection library in the NCBI C Toolkit);
2. Upper-level library written in C++ and using C++ streams.

Functionality of the library includes:

- SOCK interface (sockets), which works interchangeable on most UNIX flavors, MS-Windows, and Mac;

- SERV interface, which provides mapping of symbolic service names into server addresses;
- CONN interface, which allows to create a *connection*, the special object capable to do read, write etc. I/O operations;
- C++ streams built on top of CONN interface.

Note: The most lower-level SOCK interface is not covered in this document. Well-commented API can be found in *connect/ncbi_socket.h*.

Connections

There are 3 simple types of connections: *socket*, *file* and *http*; and one hybrid type, *service* connection.

A connection is created with a call to ***CONN_Create()***, declared in *connect/ncbi_connection.h*, and returned by a pointer to ***CONN*** passed as a second argument:

```
CONN conn; /* connection handle */
EIO_Status status = CONN_Create(connector, &conn);
```

The first argument of this function is a handle of a so-called *connector*, a special object implementing functionality of the connection being built. Above, for each type of connection there is a special connector in the library. For each connector, one or more "constructors" defined, each returning the connector's handle. Connectors' constructors are defined in individual header files, like *connect/ncbi_socket_connector.h*, *connect/ncbi_http_connector.h*, *connect/ncbi_service_connector.h* etc. ***CONN_Create()***resets all timeouts to the default value CONN_DEFAULT_TIMEOUT.

After successful creation with ***CONN_Create()***, the following calls from CONN API *connect/ncbi_connection.h* become available. All calls (except ***CONN_GetTimeout()*** and ***CONN_GetType()***) return I/O completion status of type ***EIO_Status***. Normal completion has code eIO_Success.

- **CONN_Read** (CONN conn, void* buf, size_t bufsize, size_t* n_read, EIO_ReadMethod how); Read or peek data, depending on read method how, up to bufsize bytes from connection to specified buffer buf, return (via pointer argument n_read) the number of actually read bytes. The last argument how can be one of the following:
  - eIO_ReadPlain - to read data in a regular way, that is extracting data from the connection,
  - eIO_ReadPeek - to peek data from the connection, i.e. the next read operation will see the data again,

- `eIO_ReadPersist` - to read exactly (not less than) `bufsize` bytes or until an error condition occurs.

Return value other than `eIO_Success` means trouble. Specifically, return value `eIO_Timeout` indicates that the operation could not be completed within the preset amount of time; but some data may, however, be available in the buffer (e.g. in case of persistent reading, with `eIO_ReadPersist`), and this is actually the case for any return code.

- **`CONN_Write`** `(CONN conn, const void* buf, size_t bufsize, size_t* n_written);` Write the specified number of bytes `bufsize` from the buffer `buf` to the connection. Return (via `n_written`) the number of actually written data, and completion code as a return value.

- **`CONN_Flush`** `(CONN conn);` Flush internal output queue, if this is supported by the current connection type.

- **`CONN_SetTimeout`** `(CONN conn, EIO_Event action, const STimeout* timeout)`; Set the timeout on the specified I/O action, `eIO_Read`, `eIO_Write`, `eIO_ReadWrite`, `eIO_Open`, and `eIO_Close`. The latter 2 actions are used in a phase of opening and closing the link, respectively: if connection cannot be established (closed) within the specified period, `eIO_Timeout` would result. `eIO_Timeout` results if reading/writing could not be completed within specified time range, correspondingly. A timeout can be passed as the `NULL`-pointer. This special case denotes an infinite value for that timeout. Also, a special value `CONN_DEFAULT_TIMEOUT` may be used for any timeout. This value specifies the timeout set by default for the current connection type.

- a**`CONN_GetTimeout`** `(CONN conn, EIO_Event action);` Obtain (via return value of type ***const STimeout****) timeouts set by ***>CONN_SetTimeout()*** routine, or active by default (i.e. special value `CONN_DEFAULT_TIMEOUT`). **Caution**: Returned pointer valid only for the time the connection handle is valid, i.e. up to a call to ***CONN_Close()***.

- **`CONN_ReInit`** `(CONN conn, CONNECTOR replacement);` This function allows to clear current contents of a connection, and "immerse" a new connector into it. Previous connector (if any) is closed first (if open), then gets destroyed, and thus must not be referenced again in the program. As a special case, new connector can be the same connector, which is currently active within the connection. It this case, the connector is not destroyed, instead it will be effectively re-opened. If connector passed as `NULL`, then the `conn` handle is kept existing but unusable (the old connector closed and destroyed), and can be ***CONN_ReInit()***ed later. None of the timeouts are touched by this call.

- **`CONN_Wait`** `(CONN conn, EIO_Event event, const STimeout* timeout);` Suspend the program until connection is ready to perform reading (`event =eIO_Read`) or writing (`event = eIO_Write`), or until timeout (if non-`NULL`) expired. If timeout is passed as `NULL`, then the wait time is infinite.

- **`CONN_Status`** `(CONN conn, EIO_Event direction);` Provide the information about recent low-level data exchange in the link. Operation direction has to be specified as either `eIO_Read` or `eIO_Write`. The necessity of this call arises from the fact that sometimes return value of a `CONN` API function does not really tell that the problem has been detected: suppose, the user peeks data into a 100-byte buffer and gets 10 bytes. Return status `eIO_Success` signals that those 10 bytes were found in the connection okay. But how to know whether the end-of-file condition occurred during last operation? It is where ***CONN_Status()*** comes handy. When inquired about read operation, return value `eIO_Closed` denotes that `EOF` was actually hit while making the peek, and those 10 bytes are in fact the only data left untaken, no more are expected to come.

- **`CONN_Close`** `(CONN conn);` Close the connection by closing the link (if open), deleting underlying connector(s) (if any) and the connection itself. Regardless of the return status (which may indicate certain problems), the connection handle becomes invalid and cannot be used in the program again.

- **`CONN_GetType`** `(CONN conn);` Return character string (null-terminated), verbally representing the current connection type, like *"HTTP"*, *"SOCKET"*, *"SERVICE/HTTP"* etc. Unknown connection type gets returned as `NULL`.

- **`CONN_SetCallback`** `(CONN conn, ECONN_Callback type, const SCONN_Callback* new_cb, SCONN_Callback* old_cb);` Set user callback function to be called upon an event specified by callback `type`. The old callback (if any) gets returned via passed pointer `old_cb` (if not `NULL`). Callback structure ***SCONN_Callback*** has the following fields: callback function ***func*** and *void* data*. Callback function `func` should have the following prototype:

  ```
  typedef void (*FConnCallback)(CONN conn, ECONN_Callback type, void* data);
  ```

  When called, both `type` of callback and `data` pointer are supplied. The only callback type defined as of writing of this note is `eCONN_OnClose`. Callback function is always called prior to the event to happen, e.g. close callback is called when the connection is about to close.

Note: There is no means to "open" a connection: it is done automatically when actually needed, and in most cases at the first I/O operation. But forming of actual link between source and destination can be postponed even longer. These details are hidden and made transparent to the connection's user. The connection is seen as a two-way communication channel, which is clear to use right away after a call to ***CONN_Create()***.

Note: If for some reason ***CONN_Create()*** failed to create a connection (return code differs from `eIO_Success`), then the connector passed to this function is left intact; that is, its handle can be used again. Otherwise, if connection is created successfully, the passed connector handle becomes invalid, and cannot be referenced anywhere else throughout the program (with one, however, exception: it may be used as a replacing connector in a call to `CONN_ReInit()` for the same connection).

Note: There are no public connectors' "destructors". Connector successfully put into connection is deleted automatically along with that connection by `CONN_Close()`, or explicitly with a call to `CONN_ReInit()` provided that replacing connector is `NULL` or different from the original.

Socket Connector

Constructors are defined in:

```
#include <connect/ncbi_socket_connector.h>
```

Socket connection based on the socket connector brings almost direct access to the `SOCK` API. It allows the user to create a peer-to-peer data channel between two programs, which could be located anywhere on the Internet.

In order to create the socket connection the user has to create a socket connector first, then pass it to `CONN_Create()`, as in the following example:

```
#include <connect/ncbi_socket_connector.h>
#include <connect/ncbi_connection.h>

#define MAX_TRY 3 /* Try to connect this many times before giving up */

unsigned short port = 1234;
CONNECTOR socket_connector = SOCK_CreateConnector("host.foo.com", port, MAX_TRY);

if (!socket_connector)
    fprintf(stderr, "Cannot create SOCKET connector");
else {
    CONN conn;

    if (CONN_Create(socket_connector, &conn) != eIO_Success)
        fprintf(stderr, "CONN_Create failed");
    else {
        /* Connection created ok, use CONN_... function */
        /* to access the network */
        ...
```

```
        CONN_Close(conn);
    }
}
```

A variant form of this connector's constructor, `SOCK_CreateConnectorEx()`, takes three more arguments: a pointer to data (of type ***void****), data size (bytes) to specify the data to be sent as soon as the link has been established, and flags, which presently can only be used to turn debugging information on.

`CONN` library defines two more constructors, which build SOCKET connectors on top of existing SOCK objects: ***SOCK_CreateConnectorOnTop()*** and ***SOCK_CreateConnectorOnTopEx()***, the description of which is intentionally omitted here as SOCK is not discussed either. Please refer to description in the Tookit code.

File Connector

Constructors defined in:

```
#include <connect/ncbi_file_connector.h>

CONNECTOR file_connector = FILE_CreateConnector("InFile", "OutFile");
```

This connector could be used for both reading and writing files, when input goes from one file, and output goes to another file. (This differs from normal file I/O when a single handle is used to access only one file, but rather resembles data exchange via socket.)

Extended variant of this connector's constructor, ***FILE_CreateConnectorEx()*** takes an additional argument, pointer to a structure of type ***SFileConnAttr*** describing file connector attributes, like initial read position to start from in the input file, open mode for the output file (append `eFCM_Append`, truncate `eFCM_Truncate`, or seek `eFCM_Seek` to start writing at a specified file position), and the position in the output file, which is used in *seek open mode*. Attribute pointer passed as `NULL` is equivalent to a call to ***FILE_CreateConnector()***, which reads from the very beginning of the input file, and entirely overwrites the output file (if any) implicitly using `eFCM_Truncate`.

HTTP Connector

Constructors defined in:

```
#include <connect/ncbi_http_connector.h>
```

The simplest form of this connector's constructor takes 3 parameters:

```
extern CONNECTOR HTTP_CreateConnector
(const SConnNetInfo* info,
const char* user_header,
THCC_Flags flags );
```

a pointer to network information structure (can be NULL), a pointer to a custom HTTP tag-value(s) so called user-header, and bitmask of various flags. The user-header has to be in the form "HTTP-Tag: Tag-value\r\n", or even multiple tag-values delimited and terminated by *"\r\n"*. If specified, `user_header` parameter overrides the corresponding field in `info`.

Network information structure (from *connect/ncbi_connutil.h*) defines parameters of the connection point, where the HTTP server is running. **Note**: Not all parameters of the structure depicted below apply to this connector.

```
/* Network connection related configurable info struct
*/
typedef struct {
    char                client_host[64];       /* effective client host-
name */
    char                host[64];              /* host to connect to */
    unsigned short      port;                  /* port to connect to, host
byte order */
    char                path[1024];            /* service: path(e.g. to a
CGI script) */
    char                args[1024];            /* service: args(e.g. for a
CGI script) */
    EReqMethod          req_method;            /* method to use in the
request */
    STimeout            timeout;               /* I/O timeout */
    unsigned            int max_try;           /* max. # of attempts to
establish conn */
    char                http_proxy_host[64];   /* hostname of HTTP proxy
server */
    unsigned short      http_proxy_port;       /* port # of HTTP proxy
server */
    char                proxy_host[64];        /* host of CERN-like fire-
wall proxy srv */
    EDebugPrintout      debug_printout;        /* printout some debug info
*/
    int/*bool*/         stateless;             /* to connect in HTTP-like
fashion only */
    int/*bool*/         firewall;              /* to use firewall/relay in
connects */
    int/*bool*/         lb_disable;            /* to disable local load-
balancing */ const
    char*               http_user_header;      /* user header to add to
HTTP request */

    /* the following field(s) are for the internal use only! */
    int/*bool*/         http_proxy_adjusted;
} SConnNetInfo;
```

Caution: Unlike other "static fields" of this structure, `http_user_header` (if non-`NULL`) is assumed to be dynamically allocated on the heap (via a call to ***malloc***, ***calloc*** or related function, like ***strdup***).

While the user can create and fill out this structure via field-by-field assignments, there is however a better, easier, much safer and configurable way (and interface defined in *connect/ncbi_connutil.h*) to deal with this structure:

- **`ConnNetInfo_Create`** `(const char* service)` Create and return a pointer to new ***SConnNetInfo*** structure, filled with parameters specific either for a named `service` or by default if `service` specified as `NULL` (most likely the case for ordinary HTTP connections). Parameters for the structure are taken from (in order of precedence):
  - Environment variables of the form `<service>_CONN_<name>`, where `name` is the name of the field;
  - Service-specific registry section (see below, Note about the registry) named *[service]* using the key CONN_<name>;
  - Environment variable of the form `CONN_<name>`;
  - Registry section named *[CONN]* using *name* as a key;
  - And finally, default value is applied, if none of the above resulted in a successful match.

  Search for the keys in both environment and registry is not case-sensitive; but the values of the keys are case-sensitive (except for enumerated types and boolean values, which can be of any - even mixed - case). Boolean fields accept *1*, *"YES"* and *"TRUE"* as *true* values, all other values are treated as *false*. In addition to a floating point number treated as a number of seconds, `timeout` can accept (case-insensitively) keyword *"INFINITE"*. **Note**: The first 2 steps in the above sequence are skipped if `service` name is passed as `NULL`. **Caution**: The library does not provide reasonable default values for `path` and `args` when the structure is used for HTTP connectors.
- **`ConnNetInfo_Destroy`** `(SConnNetInfo* info)` Delete and free the info structure via passed pointer (note that the HTTP user header `http_user_header` is freed, too).

- **`ConnNetInfo_SetUserHeader`** `(SConnNetInfo* info, const char* new_user_header)` Set the new HTTP user header (freeing the previous one if any) by cloning the passed string argument and storing it in `http_user_header` field. `New_user_header` passed as `NULL` resets the field.
- **`ConnNetInfo_Clone`** `(SConnNetInfo* info)` Create and return pointer to a new ***SConnNetInfo*** structure, which is an exact copy of the passed structure. This function is aware of dynamic nature of the HTTP user header field.

Note about the registry. The registry used by `connect` library is separate from ***CNcbiRegistry*** class. To learn more about the difference, and how to use both objects together in a single program, please follow this link.

The following fields of ***SConnNetInfo*** pertain to the HTTP connector: `client_host`, `host`, `port`, `path`, `args`, `req_method` (can be one of *"GET"*, *"POST"*, and *"ANY"*), `timeout`, `max_try` (analog of maximal try parameter for the socket connector), `http_proxy_host`, `http_proxy_port`, `debug_printout` (values are *"NONE"* to disable any trace printout of the connection data, *"SOME"* to enable printing of ***SConnNetInfo*** structure before each connection attempt, and *"DATA"* to print both headers and data of the HTTP packets in addition to dumps of ***SConnNetInfo*** structures). Values of other fields are ignored.

Argument `flags` in the HTTP connector's constructor is a bitwise *OR* of the following values:

- `fHCC_AutoReconnect` Allow multiple request/reply HTTP transactions. (Otherwise by default, only one request/reply is allowed.)
- `fHCC_SureFlush` Always flush a request (maybe solely consisting of HTTP header with no body at all) down to the HTTP server before preforming any read or close operations.
- `fHCC_KeepHeader` By default, HTTP connection sorts out the HTTP header and parses HTTP errors (if any). Thus, normally reading from the connection returns data from the HTTP body only. The flag disables this feature, and the HTTP header is not parsed but instead passed 'as is' to the application on a call to ***CONN_Read()***.
- `fHCC_UrlDecodeInput` Decode input data passed in HTTP body from the HTTP server.
- `fHCC_UrlEncodeOutput` Encode output data passed in HTTP body to the HTTP server.
- `fHCC_UrlCodec` Perform both encoding and encoding (`fHCC_UrlDecodeInput | fHCC_UrlEncodeOutput`).

- `fHCC_UrlEncodeArgs` Encode URL if it contains special characters like '+'. By default, the arguments are passed 'as is' (exactly as taken from ***SConnNetInfo***).
- `fHCC_DropUnread` Drop unread data, which might exist in connection, before making another request/reply HTTP shot. Normally, the connection first tries to read out the data from the HTTP server entirely, until `EOF`, and store them in the internal buffer even if either application did not requested the data for reading, or the data were read only partially, so that the next read operation will see the data.
- `fHCC_NoUpread` Do not attempt to empty incoming data channel into a temporary intermediate buffer while writing to the outgoing data channel. By default, writing always makes checks that incoming data are available for reading, and those data are extracted and stored in buffer. This approach allows to avoid I/O deadlock, when writing creates a backward stream of data, which if unread blocks the connection entirely.

The HTTP connection will be established using the following URL - http://host:port/path?args

Note that `path` has to have a leading slash "/" as the very first character, that is, only "http://" and "?" are added by the connector, all other characters appear exactly as specified (but maybe encoded with `fHCC_UrlEncodeArgs`). The question mark does not appear if the URL has no arguments.

More elaborate form of the HTTP connector's constructor has the following prototype:

```
typedef int/*bool*/ (*FHttpParseHTTPHeader)
(const char*   http_header,
void* adjust_ data,
int/*bool*/   server_error);

typedef int/*bool*/ (*FHttpAdjustInfo)
(SConnNetInfo* info,
void*         adjust_data,
unsigned int  n_failed);

typedef void (*FHttpAdjustCleanup)
(void* adjust_data
);

extern CONNECTOR HTTP_CreateConnectorEx
(const SConnNetInfo*  net_info,
THCC_Flags           flags,
FHttpParseHTTPHeader parse_http_hdr, /* may be NULL, then no addtl.parsing */
FHttpAdjustInfo      adjust_info,    /* may be NULL, then no adjustments */
void*                adjust_data,    /* for "adjust_info"& "adjust_cleanup"*/
FHttpAdjustCleanup   adjust_cleanup  /* may be NULL */ );
```

This form is assumed to be rarely used by the users directly, but it provides rich access to the internal management of HTTP connections.

The first two arguments are identical to their counterparts in the arguments number one and three of ***HTTP_CreateConnector()***. HTTP user header field (if any) is taken directly from `http_user_header` field of ***SConnNetInfo***, pointer to which is passed as `net_info` (which in turn can be passed as `NULL` meaning to use the environment, registry and defaults as described above).

The third parameter specifies a callback to be activated to parse the HTTP reply header (passed as a single string, with CR-LF - carriage return/line feed - characters incorporated to divide header lines). This callback also gets some custom data `adjust_data` as supplied in the fifth argument of the connector's constructor, and a boolean value *true* if parsed response code from the server was not okay. The callback can return *false* (zero), which is considered the same way as having an error from the HTTP server. However, pre-parsed error condition (passed in `server_error`) retains even if the return value of the callback is *true*, that is the callback is unable to "fix" the error code from the server. This callback is **not called** if`fHCC_KeepHeader` is set in flags.

The forth argument is a callback, which gets control when an attempt to connect to the HTTP server has failed. On entry, this callback has current ***SConnNetInfo***, which is requested to be adjusted in a faith that the connection to the HTTP server will finally succeed. That is, the callback can change anything in the info structure, and the modified structure will be kept for all further connection attempts, until changed by this callback again. The number (starting from 1) of successive failed attempts is given in the last callback's argument. The callback return value *true* (non-zero) means successful adjustment. Return value *false* (zero) stops connection attempts and returns an error to the application.

When connector is being destroyed, the custom object `adjust_data` can be destroyed in the callback, specified as the last argument of the extended constructor.

Note: Any callback may be specified as `NULL`, which means that no action is foreseen by the application, and default behavior occurs.

Service Connector

Constructors defined in:

```
#include <connect/ncbi_service_connector.h>
```

This is the most complex connector in the library. It can initiate data exchange between an application and a named NCBI service, and data link can be either wrapped in HTTP or be just a byte-stream (like in a socket). In fact, this connector sits on top of either HTTP or SOCKET connectors.

The library provides two forms of connector's constructor:

```
SERVICE_CreateConnector(const char* service_name);

SERVICE_CreateConnectorEx
(const char*           service_name,  /* The registered name of an NCBI service */
```

```
TSERV_Type             types,         /* Accepted server types, bitmask */
const SConnNetInfo*    net_info,      /* Connection parameters */
const SSERVICE_Extra*  params /*         Addtl set of parameters, may be NULL */ );
```

The first form is equivalent to *SERVICE_CreateConnectorEx(service_name, fSERV_Any, 0, 0)*.A named NCBI service is a CGI program or standalone server (can be one of two supported types), which runs at the NCBI site, and accessible by the outside world. Special dispatcher (which runs on the NCBI Web-servers) allows automatic switching to the appropriate server without having the client to know a priori the connection point. That is, the client just uses the main entry gate of the NCBI Web (usually, www.ncbi.nlm.nih.gov) with a request to have a service "*service_name*", and depending on the service availability, the request will be either honored (by switching and routing the client to the machine actually running the server: clicking on the previous link should bring you to a page containing "name=value" message, obtained from the special bouncing service as a result of the form submission), rejected, or declined. To the client, the entire process of dispatching is seen as completely transparent (for example, try clicking several times on either of the latter two links and see that the error replies are actually sent from different hosts, so is the successful processing of the first link done by one of several hosts running the bouncing service).

Dispatching protocol per se is implemented on top of the HTTP protocol, and is parsed by a CGI program `dispd.cgi` (or another dispatching CGI), which is available on the NCBI Web. On every server running the named services, another program, called load-balancing daemon (`lbsmd`), is executing. This daemon supports having the same service running on different machines, and allows to choose among them the one machine, which is less loaded. When `dispd.cgi` receives a request for a named service, it first consults the load-balancing table, which is broadcasted by each load-balancing daemon, and re-assembled in a network-wide form on each server. When the corresponding server is found, the client request can be passed, or a dedicated connection to the server can be established. The dispatching is made such a way that it can be also used directly from the Internet browsers.

The named service facility uses the following distinction of server types:

- HTTP servers, which are usually CGI programs:
  - ***HTTP_GET*** servers are those accepting requests only using HTTP `GET` method.
  - ***HTTP_POST*** servers are those accepting requests only using HTTP `POST` method.
  - ***HTTP*** servers are those accepting both of either `GET` or `POST` methods.

- ***NCBID*** servers are those run by a special CGI engine, called `ncbid.cgi`, a configurable program (now integrated within `dispd.cgi` itself), which can convert byte-stream output from another program (server) started by the request from dispatcher to an HTTP-compliant reply (that is a packet having both HTTP header and body, and suitable e.g. for Web-browsers).
- ***STANDALONE*** servers, like mailing daemons, are those listening on their own onto the network for incoming connections.
- ***FIREWALL*** servers are the special pseudo-servers, not existing in reality, but are created and used internally by the dispatcher software to indicate that only a firewall connection mode can be used to access the requested service.
- ***DNS*** servers are beyond the scope of this document cause they are to declare domain names, which used internally at NCBI site to help load-balancing based DNS lookup (see here).

Formal description of these types is given in *connect/ncbi_server_info.h*:

```
/* Server types
*/
typedef enum {
  fSERV_Ncbid      = 0x1,
  fSERV_Standalone = 0x2,
  fSERV_HttpGet    = 0x4,
  fSERV_HttpPost   = 0x8,
  fSERV_Http       = fSERV_HttpGet | fSERV_HttpPost,
  fSERV_Firewall   = 0x10,
  fSERV_Dns        = 0x20
} ESERV_Type;

#define fSERV_Any           0
#define fSERV_StatelessOnly 0x80
typedef unsigned TSERV_Type;  /* bit-wise OR of "ESERV_Type" flags */
```

The bitwise *OR* of the ***ESERV_Type*** flags can be used to restrict the search for the servers, matching the requested service name. These flags passed as argument `types` are used by the dispatcher when figuring out, which server is acceptable for the client. Special value *0* (or, better `fSERV_Any`) states no such preference whatsoever. Special bit-value `fSERV_StatelessOnly` set together with other bits or just alone specifies that the servers should be of stateless (HTTP-alike) type only, and it is the client which is responsible for keeping track of the logical sequence of transactions.

The following code fragment establishes service connection to the named service *"io_bounce"*, using only stateless servers:

```
CONNECTOR c;
CONN conn;
```

```
if (!(c = SERVICE_CreateConnectorEx("io_bounce", fSERV_StatelessOnly, 0, 0)))
    fprintf(stderr, "No such service available");
else if (CONN_Create(c, &conn) != eIO_Success)
    fprintf(stderr, "Failed to create connection");
else {
    static const char buffer[] = "Data to pass to the server";
    size_t n_written;

    CONN_Write(conn, buffer, sizeof(buffer) - 1, &n_written);
    ...
}
```

The real type of the data channel can be obtained via call to *>CONN_GetType(conn)*.

Note: In the above example the client has no assumption how the data actually passed to the server. The server could be of any type in principle, even a standalone server, which was used in the request/reply mode of one-shot transactions. If necessary, such wrapping would have been made by the dispatching facility as well.

The last but one parameter of the extended constructor is the network info, described in section devoted to HTTP connector. Service connector uses all fields of this structure, except for `http_user_header`, and the following assumptions apply:

- `path` specifies the dispatcher program (defaulted to `dispd.cgi`);
- `args` specifies parameters for the requested service, this is service-specific, no defaults;
- `stateless` is used to set `fSERV_StatelessOnly` flag in the server type bitmask, if it was not set there already (convenient to modify the dispatching using environment and/or registry, if the flag is not set; yet allows to hardcode the flag at compile-time by setting it in constructor's `types` argument explicitly);
- `lb_disable` set to *true* (non-zero) means to always use remote dispatcher (via network connection) even if locally running load-balancing daemon is available (by default, local load-balancing deamon consulted first to resolve the name of the service);
- `firewall` set to *true* (non-zero) disables the direct connection to the service. Instead, either a connection to a proxy firewall daemon (fwdaemon), running at the NCBI site, is initiated to pass the data in stream mode, or data get relayed via dispatcher if stateless server is used;
- `http_user_header` ignored (asserted to be `NULL` in debug compilation mode).

As with HTTP connector, if network information structure is specified as `NULL`, default values are obtained as described above, as with the call to *ConnNetInfo_Create(service_name)*.

Normally the last parameter of ***SERVICE_CreateConnectorEx()*** is left `NULL`, which sets all additional parameters to their default values. Among others, this includes default procedure of choosing an appropriate server when the connector is looking for a mapping of the service name into a server address. To see how this parameter can be used to change the mapping procedure please refer to a later section.

Library provides additional interface to named service mapper, which can be found in *connect/ncbi_service.h*.

Note: Requesting `fSERV_Firewall` in the `types` parameter effectively selects firewall mode regardless of the network parameters, loaded via ***SConnNetInfo*** structure.

Debugging Tools and Troubleshooting

Each connector (except for ***FILE*** connector) provides means to view data flow in the connection. In case of ***SOCKET*** connector debugging information can be turned on by the last argument in ***SOCK_CreateConnectorEx()***, or by using global routine ***SOCK_SetDataLoggingAPI()*** (declared in *connect/ncbi_socket.h*) **Note**: In the latter case every socket (including sockets implicitly used by other connectors like ***HTTP*** or ***SERVICE***) will generate debug printouts.

In case of ***HTTP*** or ***SERVICE*** connectors, which employ ***SConnNetInfo***, debugging can be activated directly from the environment by setting `CONN_DEBUG_PRINTOUT` to *TRUE* or *SOME*. Similarly, a registry key `DEBUG_PRINTOUT` with a value of either *TRUE* or *SOME* found in the section *[CONN]* would have the same effect: it turns on only logging of connection parameters each time the link gets established. When set to *ALL*, this variable (or key) also turns on debugging output on all underlying sockets ever created during the life of the connection. Value *FALSE* (default) turns debugging printouts off. Moreover, for ***SERVICE*** connector the debugging output option can be set on a per-service basis using `<service>_CONN_DEBUG_PRINTOUT` environment variables, or individual registry sections *[<service>]* and key *CONN_DEBUG_PRINTOUT* in them. **Note**: Debugging printouts can only be controlled in a described way via environment or registry if and only if ***SConnNetInfo*** is always created with the use of convenience routines.

Debugging output is always sent to the same destination, CORE log file, which is a C object shared between both C and C++ Toolkits. As said, the logger is an abstract object, i.e. it is empty and cannot produce any output if not tuned accordingly. The library defines few calls gathered in `connect/ncbi_util.h` which allow the logger to go via ***FILE*** file pointer, like `stderr`: ***CORE_SetLOGFILE()*** as e.g. shown in the example *test_ncbi_service_connector.c*, or to be a regular file on disk. Moreover, both Toolkits define interfaces to deal with registries, loggers and locks that use native objects of each toolkit and use them as replacements for corresponding abstract layer's objects.

There is a common problem reported several times and actually concerning network configuration rather than misbehavior of the library. If a test program, which connects to a named NCBI service, is not getting anything back from the NCBI site, one first has to check whether there is a transparent proxying/caching in between the host and NCBI. As the service dispatching is implemented on top of ordinary HTTP protocol, the transparent proxying may latch unsuccessful service searches (which could happen and may not indicate a real problem) as error responses from

the NCBI server. Afterwards, instead of actually connecting to NCBI, the proxy returns those cached errors (or sometimes just an empty document), which breaks the service dispatcher code. In most cases there are configurable ways to exclude certain URLs from proxying and caching, and they are subject for discussion with a local network administrator.

There is another tip: Make sure that all custom HTTP header tags (as passed, for example, in SConnNetInfo::user_header field) do have "\r\n" as tag separators (including the last tag). Many proxy servers (including transparent proxies, which usually the user is not even aware of) are known to be sensitive to that each and every HTTP tag is closed by "\r\n" (and not by a single "\n" character). Otherwise the HTTP packet would be treated as lame and gets discarded.

C++ Connection Streams

Using connections and connectors (via the entirely procedural approach) in C++ programs overkills the power of the language. Therefore, we provide C++ users with the stream classes, all derived from standard ***iostream*** class, and as a result, which can be used with all famous stream I/O operators, manipulators etc.

The declarations of the stream's constructors can be found in *connect/ncbi_conn_stream.hpp*. We tried to keep the same number and order of constructor's parameters, as they appear in the corresponding connector's constructors in C.

The code below is a C++-style example from the previous section devoted to the service connector:

```
#include <connect/ncbi_conn_stream.hpp>

    try {
        CConn_HttpStream
        ios("io_bounce", fSERV_StatelessOnly, 0); ios << "Data to be passed to the
server";
    } STD_CATCH_ALL("Connection problem");

    ...
```

Note: Stream constructor may throw an exception if, for instance, the requested service is not found, or other kind of problem arose. To see the actual reason, we used standard toolkit macro `STD_CATCH_ALL()`, which prints the message and problem description into the log file (`cerr`, by default).

Service mapping API

The API defined in *connect/ncbi_service.h* maps required sevice name into server address. Internally, the mapping is done either directly or indirectly by means of load-balancing daemon, running at NCBI site. For the client, the mapping is seen as a reading from an iterator created by a call to ***SERV_Open()*** like in the following fragment (for more examples please refer to the test program *test_ncbi_disp.c*):

```
#include <connect/ncbi_service.h>

    SERV_ITER iter = SERV_Open("my_service", fSERV_Any, SERV_ANYHOST, 0);
    int n = 0;

    if (iter != 0) {
        SSERV_Info* info = SERV_GetNextInfo(iter);
        while (info != 0) {
            char* str = SERV_WriteInfo(info);

            printf("Server = `%s'\n", str);
            free(str);
            n++;
        }
        SERV_Close(iter);
    }
    if (!iter || !n)
        printf("Service not found\n");
```

Note: Non-`NULL` iterator returned from ***SERV_Open()*****does not** yet guarantee that the service is available, whereas `NULL` iterator definitely means that the service does not exist.

As shown in the above example, loop over reading from the iterator results in the sequence of successive structures (none of which is to be freed by the program!) that along with conversion functions ***SERV_ReadInfo()***, ***SERV_WriteInfo()*** and others are defined in *connect/ncbi_server_info.h*. Structure ***SSERV_Info*** describes a server that implements requested service. `NULL` gets returned when no more servers (if any) could be found. The iterator returns servers in the order the load-balancing algorithm arrange them. Each server has a rating, and the larger the rating the better the chance for the server to be chosen first.

Note: Servers returned from the iterator are all of the requested type, with only one exception: they can include servers of type `fSERV_Firewall` even if this type was not explicitly requested. So the application must sort these servers out, if not interested in them. But if `fSERV_Firewall` is set in the types, then the search is done for whichever else types requested, and with the assumption that the client has chosen firewall connection mode, regardless of network parameters supplied in ***SConnNetInfo***, or read out from either registry or environment.

Note: Search for servers of type `fSERV_Dns` is not inclusive with `fSERV_Any` specified as server type. That is, servers of type DNS are only returned if specifically requested in the server mask at the time the iterator was opened.

There is a simplified version of ***SERV_Open()***, called ***SERV_OpenSimple()***, as well as an advanced version, called ***SERV_OpenEx()***. The former takes only one argument, the service name. The latter takes two more arguments, which describe the set of servers **not** to be seen from the iterator (excluded server descriptors).

There is also an advanced version of ***SERV_GetNextInfo()***, called ***SERV_GetNextInfoEx()***, which via its second argument allows to get many host parameters, among which is a so called host environment, a *"\0"*-terminated string, consisting of set of lines separated by *"\n"* characters,

and specified in the configuration file of load-balancing daemon of the host, where the returned server was found. The typical line within the set has a form *"name=value"* and resembles very much the shell environment, where its name comes from. The host environment could be very handy for passing additional information to applications if the host has some limitations or requires special handling should the server be selected and used on this host. Example below should give an idea. At the time of writing, getting the host information is only implemented when the service is obtained via direct access to the load-balancing daemon. Unlike returned server descriptors, the returned host information handle is not a constant object and must be explicitly freed by the application when no longer needed. All operations (getter methods) that are defined on the host information handle are declared in *connect/ncbi_host_info.h*. If the server descriptor was obtained using dispatching CGI (indirect dispatching), then the host information handle is always returned as `NULL` (no host information available).

The back end of the service mapping API is split into 2 independent parts: *direct* access to >LBSMD, if the one is both available on the current host and is not disabled by parameter -->`lb_disable` at the iterator opening. If `LBSMD` is either unavailable or disabled, the second (*indirect*) part of the back-end API is used, which involves connection to dispatching CGI, which in turn connects to `LBSMD` in order to carry out the request. Attempt to use the CGI is only done, if `net_info` argument is provided non-`NULL` in the calls to ***SERV_Open()*** or ***SERV_OpenEx()***. **Note**: In call to ***SERV_OpenSimple()***, `net_info` gets created internally before upcall to ***SERV_Open()*** and thus CGI dispatching is likely to happen, unless either `net_info` could not be constructed from the environment, or environment varaible `CONN_LB_DISABLE` (or service-specific one, or either of corresponding registry keys) is set to *TRUE*. **Note**: In the above conditions, the network service name resolution is also undertaken if service name could not be resolved (due to service inexistence or other error) with the use of locally found `LBSMD`.

The following code example uses both service connector and the service mapping API to access certain service using an alternate way (other than connector's default) of choosing appropriate servers. By default, service connector opens an internal service iterator and then tries to connect to the next server, which ***SERV_GetNextInfo()*** returns when given the iterator. That is, the server with higher rate is tried first. If the user provides a pointer to structure ***SSERVICE_Extra*** as the last parameter of the connector's constructor, then the user-supplied routine (if any) can be called instead in order to obtain the next server. The routine is also given a supplemental custom argument `data` taken from ***SSERVICE_Extra***. The (intentionally simplified) example below tries to create connector to an imaginary service *"my_service"* in restriction that the server has additionally to have a certain (user-specified) database present. The database name is taken from LBSMD host environment keyed *"my_service_env"*, the first word of which is assumed to be the name.

```
#include <connect/ncbi_service_connector.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define ENV_DB_KEY "my_service_env="

/* This routine gets called when connector is about to be destructed.
*/
static void s_CleanupData(void* data)
{
    free(data); /* we kept database name there */
}

/* This routine gets called on each internal close of the connector
* (which may be followed by a subsequent internal open).
*/
static void s_Reset(void* data)
{
    /* just see that reset happens by printing DB name */
    printf("Connection reset, DB: %s\n", (char*) data);
}

/* 'Iter' is an internal service iterator used by service connector; it must
* remain open.
* 'Data' is what we supplied among extra-parameters in connector's constructor.
*/
static const SSERV_Info* s_GetNextInfo(SERV_ITER iter, void* data)
{
    const char* db_name = (const char*) data;
    size_t len = strlen(db_name);
    SSERV_Info* info;
    HOST_INFO hinfo;
    int reset = 0;

    for (;;) {
        while ((info = SERV_GetNextInfoEx(iter, &hinfo)) != 0) {
            const char* env = HINFO_Environment(hinfo);
            const char* c;
            for (c = env; c; c = strchr(c, '\n')) {
                if (strncmp(c == env ? c : ++c, ENV_DB_KEY,
                            sizeof(ENV_DB_KEY)-1) == 0) {
                    /* Our keyword has been detected in environment */
                    /* for this host */
                    c += sizeof(ENV_DB_KEY) - 1;
                    while (*c && isspace(*c))
                        c++;
                    if (strncmp(c, db_name, len) == 0 && !isalnum(c + len)) {
                        /* Database match */
                        free(hinfo); /* must be freed explicitly */
                        return info;
                    }
```

```
                }
            }
            if (hinfo)
                free(hinfo); /* must be freed explicitly */
        }
        if (reset)
            break; /* coming to reset 2 times in a row means no server fit */
        SERV_Reset(iter);
        reset = 1;
    }
    return 0; /* no match found */
}

int main(int argc, char* argv[])
{
    char* db_name = strdup(argv[1]);
    SSERVICE_Extra params;
    CONNECTOR c;
    CONN conn;

    memset(&params, 0, sizeof(params));
    params.data = db_name; /* custom data, anything */
    params.reset = s_Reset; /* reset routine, may be NULL */
    params.cleanup = s_CleanupData; /* cleanup routine, may be NULL*/
    params.get_net_info = s_GetNextInfo; /* custom iterator routine */
    if (!(c = SERVICE_CreateConnectorEx("my_service",
            fSERV_Any, NULL, &params))) {
        fprintf(stderr, "Cannot create connector");
        exit(1);
    }

    if (CONN_Create(c, &conn) != eIO_Success) {
        fprintf(stderr, "Cannot create connection");
        exit(1);
    }

    /* Now we can use CONN_Read(), CONN_Write() etc to operate with
     * connection, and we are assured that the connection is made only
     * to the server on such a host which has "db_name" specified in
     * configuration file of LBSMD.
     */

    ...
    CONN_Close(conn);
    /* this also calls cleanup of user data provided in params */

   return 0;
}
```

Note: No network (indirect) mapping occurs in the above example because `net_info` is passed as `NULL` to the connector's constructor.

Threaded Server Support

This library also provides CThreadedServer, an abstract base class for multithreaded network servers. Here is a demonstration of its use. Note that this class **does not** support multiplexing traffic over a single TCP connection; rather, each thread has an individual TCP connection created when a client connects to the server's listening port.